Shaw, M. (1985). What can we specify? Questions in the domains of software specifications. Proceedings of the Third International Workshop on Software Specification and Design, IEEE Computer Society, pp. 214-215.

What Can We Specify? Issues in the Domains of Software Specifications

Mary Shaw

Software Engineering Institute
and
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract: Formal specifications customarily deal exclusively with the domain of functional properties of software. However, other domains are of interest to software designers and developers. Two particular areas of concern for practical software development are not yet well-served by formal specifications. This note raises issues about how those areas might be better served.

Current techniques for formal specification are primarily concerned with functional correctness -- that is, with the computational properties of programs. Considerable attention has been devoted to this issue, and the results have been of practical as well as theoretical significance.

Functional correctness is clearly important: programs must certainly perform their intended functions. To be useful in practice, however, programs must satisfy other requirements. Such requirements include execution within time and space bounds that can be supported by available hardware and consistent treatment of families of related software components. These requirements are often described in informal or imprecise requirement documents. Although these properties are not a part of the functionality of a program, I believe that they can and should be subjected to formal specification and proof.

My interest is in the use of formal specifications for practical software development. In this setting, some issues arise that are not handled well by classical formal specification techniques. These issues include the specification of properties of programs other than computational functionality and the definition of families of related, but not identical, software components. I will consider each of these in turn.

Properties Beyond Computational Functionality

Many properties other than functional correctness are of concern to practical software designers. Additional properties of interest include performance (e.g., time and space costs), mathematical accuracy, reliability, synchronization, and execution patterns (e.g., working set size).

The extension of formal specification techniques to include these additional properties should be done in such a way as to preserve the desirable characteristics of functional specifications. In particular:

- The generalized specifications should mesh with the abstract modular structure of the program; in this way they can be developed and studied along with the functional specifications.
- The specification methods should allow as much or as little precision as may be appropriate; excessively precise specifications can be expensive to process and can constrain future evolution of the software.
- The specifications should be both precise and mathematically tractable; formal analysis is not always feasible, but it is more reliable than informal, especially prose, arguments.
- It should not be necessary to develop a new specification methodology for each additional property of interest; it is extremely desirable to extend existing methods to new properties instead of developing new methods whenever possible.

A certain amount of work on formal specifications of these "extra-functional" properties has already been done. Areas that have received attention include execution time requirements[9] and the ways time specifications interact with functional specifications[1]; security properties of software[6,7,10]; reliability[4,11]; and communication protocols[5].

Families of Software Components

For many years, abstraction techniques have been used to cope with the complexity of programs. Generally, abstraction techniques emphasize selected detail and suppress irrelevant detail, thereby directing attention to the problem at hand. As a side effect, abstract definitions describe collections of possible code sequences rather than single code sequences.

As time has passed, the size of the program units that can be described in this way and the kinds of variability that can be accomodated has increased. At present, the most general abstract specification tools are generic definitions [3] and certain systems for type inheritance ("subclassing").

However, our informal understanding of groups of related software components still outstrips our ability to specify such components formally. For example, we can specify and implement abstract data types for *tree* and *linked list*, but we have no good way to combine these to obtain a *threaded tree* in which each node participates in both an instantiation of *tree* and an instantiation of *linked list*. For another example, we can specify and implement a record type for which we construct tables, search the tables, insert elements, etc -- but we cannot write one definition that specifies tables of records that differ in the number and types of fields, the position and name of the key field, etc. and can be instantiated for a variety of record types.

There is a serious need for systems that allow generation of a family of related software components from a single definition that captures the family characteristics and is tailored to specific applications with the addition of information that binds the variable aspects of the definition.

This issue is related to the question of maintaining intellectual control over the development of large software systems that exist in many versions (through time) and many configurations (concurrently) [2,8]. In many cases, different versions of a module are related as described above, and better control might be exercised by generating different instances from a single definition.

The Question

The question I would like to pose to this workshop is, *How can we best extend the domain of formal specification to include a variety of properties of interest to practical software developers?*

References

[1] Jon Louis Bentley and Mary Shaw.
Abstraction and Efficiency: The Interaction of Languages and Analysis.
Computer Science Research Review, Carnegie-Mellon University, 1979

[2] P.M. Cashin, M.L. Joliat, R.F. Kamel, D.M.Lasker.
Experience with a Modular Typed Language: PROTEL.
In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[3] *The Programming Language Ada Reference Manual.*
United States Department of Defense, 1983.
MIL-STD-1815A-1983.

[4] Ivor Durham and Mary Shaw.
Specifying Reliability as a Software Attribute.
Technical Report CMU-CS-82-148, Department of Computer Science, Carnegie-Mellon University, December, 1982.

[5] Donald I. Good.
Constructing Verified and Reliable Communications Processing Systems.
ACM Software Engineering Notes 2(5), October, 1977.

[6] Jonathan K. Millen.
Security Kernel Validation in Practice.
Communications of the ACM 19(5), May, 1976.

[7] Peter G. Neumann, Robert S. Boyer, Richard J. Feiertag, Karl N. Levitt and Lawrence Robinson.
A Provably Secure Operatiing System: The System, its Applications, and Proofs.
Technical Report 4332 Final Report, SRI International Project, February, 1977.

[8] Eric Emerson Schmidt.
Controlling Large Software Development in a Distributed Environment.
PhD thesis, University of California, Berkeley, 1982.

[9] Mary Shaw.
A Formal System for Specifying and Verifying Program Performance.
Technical Report CMU-CS-79-129, Carnegie-Mellon University, June, 1979.

[10] B.J. Walker, R.A. Kemmerer, and G.J. Popek.
Specification and Verification of the UCLA Unix Security Kernel.
Communications of the ACM 23(2), February, 1980.

[11] John H. Wensley, Leslie Lamport, Milton W. Green, Karl N. Levitt, P.M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock.
SIFT: Design and Analysis of a Fault-tolerant Computer for Aircraft Control.
Proceedings of the IEEE 66(10): 1240-1255, October, 1978.

The Fine Print

The opinions expressed here are those of the author and not necessarily those of Carnegie-Mellon University, the Software Engineering Institute, or the Department of Defense.